SMARTPROXY: Secure Smartphone-Assisted Login on Compromised Machines

Johannes Hoffmann, Sebastian Uellenbeck, and Thorsten Holz

Horst Görtz Institute (HGI), Ruhr-University Bochum, Germany
{firstname.secondname}@rub.de

Abstract. In modern attacks, the attacker's goal often entails illegal gathering of user credentials such as passwords or browser cookies from a compromised web browser. An attacker first compromises the computer via some kind of attack, and then uses the control over the system to steal interesting data that she can utilize for other kinds of attacks (*e. g.*, impersonation attacks). Protecting user credentials from such attacks is a challenging task, especially if we assume to not have trustworthy computer systems. While users may be inclined to trust their personal computers and smartphones, they might not invest the same confidence in the external machines of others, although they sometimes have no choice but to rely on them, *e. g.*, in their co-workers' offices.
To relieve the user from the trust he or she has to grant to these computers, we propose a privacy proxy called SMARTPROXY, running on a smartphone. The point of this proxy is that it can be accessed from untrusted or even compromised machines via a WiFi or a USB connection, so as to enable secure logins, while at the same time preventing the attacker (who is controlling the machine) from seeing crucial data like user credentials or browser cookies. SMARTPROXY is capable of handling both HTTP and HTTPS connections and uses either the smartphone's Internet connection, or the fast connection provided by the computer it is linked to. Our solution combines the security benefits of a trusted smartphone with the comfort of using a regular, yet untrusted, computer, *i. e.*, this functionality is especially appealing to those who value the use of a full-sized screen and keyboard.

Keywords: Web Security, Browser Security, Privacy Proxy, Smartphone, SSL

1 Introduction

Regardless of the passing years witnessing numerous efforts to secure software systems, we are still observing security incidents happening on a daily basis. Attackers have managed to compromise the integrity of computer systems, as techniques such as control-flow attacks on software systems or exploiting logical flaws of applications were developed [2,11,23,25]. Due to the growing complexity of today's systems, it is highly unlikely that we will have trustworthy computer systems without *any* security flaw at our disposal in the near future. We thus need to investigate how certain security properties can be guaranteed despite compromised computer systems being in operation. In this paper, we focus on the threat of stealing user credentials from web browsers. More precisely, the attacker's goal often entails illegal gathering of user credentials

such as passwords or browser cookies from an already compromised machine. Several reports on malicious software specialized in stealing credentials provides evidence for the prevalence of this threat (*e. g.*, bots and keyloggers) [9,15].

In practice, we do not only need to worry about the integrity of our personal (home or office) computer, but of each and every machine we interact with and enter our credentials into. Surfing on public Internet terminals, using computers in Internet cafes, or simply browsing the web on a arbitrary in-office or friend's computer can all bring about a number of potential threats. A golden rule of thumb for our safety is that we cannot trust any such machines, or going one step further: we have to assume that they have been compromised. Still, all things considered, protecting the credentials from attackers in such situations proves to be a major challenge: if an attacker has compromised a system, she has complete and full control over the system and can modify the settings in an arbitrary way. Furthermore, she can read all memory contents, the CPU state, and all other information pertaining to the current state of the system in question. Based on the so-obtained data, an attacker can easily reconstruct (user) credentials [12,14] as it becomes harder (if not impossible) to protect the integrity of user personal, login, and other sorts of data.

One idea for improving the integrity of credentials is to use external trusted devices and generate one-time keys (*e. g.*, RSA SecurID [8] or Yubico YubiKeys [27]). Here, even if an attacker has compromised the system, she only obtains a one-time password that cannot be reused, hence the potential harm is greatly limited. Nonetheless, this approach does not support legacy applications and requires changes on the server's side since the one-time password needs to be verified by the server. Furthermore, the compromise of RSA and Lockheed Martin in 2011 [5] has illustrated that such systems can also be periled, provided that the seed files for the authentication tokens can be accessed by a skilled attacker. A different idea is to use an external device with more computational power as a trusted intermediary supporting the authentication routine [4,17,26]. Such approaches typically either require changes on the server side or an additional server that needs to be trusted. As a result, they are typically not compatible with existing systems or trust needs to be put into another device not under the user's control.

In this paper, we introduce a system called SMARTPROXY that protects user credentials when interacting with a compromised machine. The basic idea is to use a smartphone as a special kind of *privacy proxy* as we shall tunnel all network connections of the untrusted machine over the phone and filter all sensitive data there. To be more precise, we intend the user to only enter fake information on the untrusted machine, as we will have our tool substitute this data with the real credentials on the outgoing network connections on the fly. For incoming network connections, SMARTPROXY will also replace different kinds of sensitive data (*i. e.*, cookies) with imitative information, so that a potential attacker operating from the untrusted machine cannot achieve her goals. As a result, a (semi-)trusted smartphone takes care of handling sensitive data related to the user and therefore, the confidential information never reaches the untrusted machine on which an attacker could potentially track and observe it. This is achieved through selective usage of encrypted communication channels via SSL: SMARTPROXY intercepts the phone's outgoing SSL connections and performs a Man-in-the-Middle (MitM) attack, eventually being capable of substituting fake credentials with valid ones. All of

the tool-initiated network connections to the destination server additionally use SSL, meaning that an attacker cannot intercept or eavesdrop on those channels.

Since the phone is a potential target for the attacks, we also make sure that we split the transferred information in such a way that even if an attacker compromises both the machine *and* the smartphone, she at least cannot steal all data at once. This is achieved by encrypting all private data with a user-selected key for each credential. Conclusively, our tool enables a sound and robust method of credentials' handling. We have implemented a fully working prototype of SMARTPROXY for Android-based devices. Our evaluation demonstrates that the overhead introduced by the tool is reasonable: micro-benchmarks indicate that the SSL handshake typically takes less than 42 ms on a current smartphone, while in macro-benchmarks we found a typical overhead of less than 50 percent to the load time for popular websites.

In summary, we make the following contributions:

- We introduce the concept of a smartphone-based privacy proxy that can be used to preserve credentials in spite of the presence of a compromised machine that an attacker controls.
- We have implemented a fully-working prototype and our performance benchmarks indicate that the overhead of SMARTPROXY on typical websites is moderate.

2 System Overview

In this section, we describe the attacker model which has been employed throughout the rest of this work. Furthermore, we give a brief overview of our approach to securing user credentials in face of a compromised machine and sketch the high-level idea.

2.1 Attacker Model

In the following, we assume that an attacker has compromised a computer system and thus has complete control over it. This allows her to obtain the system state's full information and enables arbitrary modifications. On a practical level, this might be the case when an attacker managed to infect a computer with some kind of malicious software that can steal credentials and/or alter the settings of a web browser. Furthermore, we assume that the attacker cannot break a public key crypto system and symmetric ciphers with reasonable parameters, *i. e.*, we assume that unless the used key is known and out in the open, the used ciphersuites cannot be broken by the attacker. As a special action, the attacker might disable our proxy for the web browser. Doing this will only result in a denial of service attack against the user, as long as the user always only enters fake credentials into the web browser.

A user who wants to utilize the web browser and log in to a given website such as Facebook or Wikipedia needs to have valid user credentials. Essentially, he has no simple means to decide whether the machine he wishes to put to work is indeed compromised or not since it is not a machine under his control (*e. g.*, a public Internet terminal or an arbitrary in-office computer). He may also simply not know how to verify the integrity of the machine in question. Overall, it is a complex problem in itself and we

assume that the user cannot efficiently assess whether the endangerment has taken place or not. Luckily, the user has as an auxiliary means of resorting to his smartphone which he can, to a certain extent, trust. We suppose for now that the smartphone has not been compromised, which implies that the user can run software on the phone without having to worry about an attacker. At the same time, we understand that the user wants to surf the Web on a computer rather than a smartphone since it provides a larger screen and a natural-sized keyboard. In addition, the computer has more computational power, as it is for example capable to render movies quickly and with ease.

Based on these assumptions, the goal of our system can be clearly indicated as enabling the user to log in to a website without vexing about his credentials. We focus on the HTTP and HTTPS protocol, connections with arbitrary protocols over sockets are not considered for now. With our tool in place, the attacker will neither acquire the login credentials (*i. e.*, username, passwords, or browser cookies) nor obtain any information about them. Note that the system cannot protect the web content on the computer as the attacker is already assumed to be able to read all information on that device. For that reason, the attacker can still read as well as modify all content accessed by the user, yet she cannot log in on the website at a later point in time as all valid user credentials are no longer there to be taken away.

In addition, we need to make sure that an attacker cannot obtain credentials by compromising the smartphone. This would imply that our system has a single point of failure. Even if the attacker can compromise both the computer *and* the smartphone at the same time, she remains unable to acquire all valid credentials. By splitting the actual credentials in a clever way (see Section 3.6), we warrant that an attacker can only observe the data which are evidently made use of in real time.

2.2 System Overview

As stated above, in order to protect user credentials from an attacker operating a compromised machine, we propose to use a smartphone as a kind of privacy proxy. As a preparatory step, the user needs to configure the web browser on the computer in such a way that it engages the smartphone as a web proxy for HTTP and HTTPS traffic. As we discuss in Section 3.2, there are quite a few different ways for a smartphone being connected to a computer, which leads us to pinpointing an easy way to set up our system. Moreover, the user needs to import a X509 V1 root certificate into the web browser, as we need to intercept SSL connections (as we will explain later on).

Once the machine and the smartphone are connected, the smartphone transparently substitutes fake credentials entered by the user on the compromised computer with valid credentials, which are then sent to the target destination the user wants to visit. The smartphone can perform this substitution by carrying out a MitM attack on the HTTPS connection: our tool intercepts the initial SSL connection attempt from the computer, establishes its own SSL session to the target web server on behalf of the computer, and transparently substitutes the fake credentials with valid ones. These steps require us to launch a fake SSL connection to the web browser on the computer, for which we spoof a HTTPS session pretending to be the targeted web server. We generate an SSL certificate that corresponds to the targeted server using the root certificate imported in the preparation phase. Thus, this certificate will be accepted by the browser and the user

can regard this connection trustworthy. We discuss the workflow of our approach and the MitM attack in more detail in Sections 3.3 and 3.4.

Moving on, the smartphone supplants sensitive information sent from the website to the browser (such as browser cookies) in a way that an attacker cannot obtain them. Our tool successfully alters information on one hand, yet grants the possibility of a web browser usage on the other. Meanwhile an attacker does not have the power to obtain any useful information from the processes taking place. More details about this substitution procedure and its security aspects are available in Section 3.5.

Effectively, we move the handling of sensitive data from the (potentially compromised) computer to the smartphone, which prevents an attacker from stealing sensitive data. De facto, this carries the problem over to another device which we need to protect: the smartphone turns into an attractive target for the attackers, as it now serves as a primary storage space for all the sensitive data. At the end, we need to be fully confident that the information is kept and retained in a way that fully prevents an attacker from accessing it, even if she manages to compromise *both* the computer and the smartphone at the same time. This can be achieved by storing and provisioning information in an encrypted manner. Decrypting this data may only occur strictly on demand, as we explain in detail in Section 3.6. The potential damage is here-limited to the actually-used credentials exclusively.

3 Implementation

We now describe the implementation of the ideas sketched above in a tool called SMARTPROXY. More specifically, we explain how the browser is able to communicate with the proxy running on a smartphone and what user interactions are required. We also clarify how the whole process of enabling secure logins on compromised machines is handled by the proxy. We conclude this section with a description of secure storage of private user data within SMARTPROXY, as it is employed to minimize the attack surface.

3.1 Software Overview

SMARTPROXY is implemented as an Android application in Java and we use the *Bouncy Castle* library to forge certificates. All services related to SSL are provided by Android's own SSL provider *AndroidOpenSSL*. Our implementation of the HTTP protocol supports both HTTP/1.0 and HTTP/1.1 (we implemented a required subset of the protocol, not the whole specification), multiple concurrent connections, SSL/TLS (for the rest of the paper we simply use the shortened term SSL), and HTTP pipelining of browser requests. The proxy software includes a graphical user interface on the smartphone, mainly utilized to start or stop the proxy and manage security-related aspects of the tool. More precisely, a user can manage trusted and forged certificates, stored cookies, and user credentials within the interface. The proxy itself listens on two different TCP ports, one for the plain HTTP traffic, and a second one for the encrypted HTTPS traffic.

One goal of the design was the minimization of required user interaction since SMARTPROXY should not become yet another complicated task to deal with at the user's end. Nevertheless, some user interaction is unavoidable for normal operation.

For the proxy to be used, it has to be connected to the computer in one of the ways enumerated in the next section. The browser on the untrusted computer then needs to be configured to employ the smartphone as proxy. This simply requires setting the smartphone's IP address and the ports as the proxy address of the browser. If it is the first time for engaging the proxy, an automatically generated X509 V1 root certificate has to be exported from the smartphone (*e. g.*, either through the SD card or by mounting the smartphone as external storage device). It shall later be imported into the web browser as a trusted root certificate destined and valid for signing other websites' certificates.

Once the setup process has been completed, any type of user interaction is required in only two additional cases. First, in the instance where our SSL trust manager is unable to verify a server certificate. In practice, this might happen in several different circumstances, *e. g.*, if the certificate is self-signed, not signed by a trusted authority, or if the certificate is for some reason invalid (*e. g.*, it has expired). Beware that an invalid certificate might equally indicate an attack as we discuss in Section 3.4. If a certificate cannot be verified, the user has to manually examine and perhaps approve the certificate in a dialog on the smartphone. This is a replication of a security model behavior as it can be found in web browsers. If a user accepts the certificate, an exception for this certificate is generated and further on it will be perceived and processed as validated. Second case of user interaction demand is of course linked to the setup of the fake credentials for any website that the user wishes to log in to securely.

In addition to the aforementioned interactions, the user is able to list and delete all stored cookies and edit credentials with their original and substituted values. SMARTPROXY also enables a user to list and delete all forged and manually trusted certifications. For example, it might at one point be necessary to issue a removal of a user-trusted certificate which has been later on proven invalid and shall therefore no longer be used to establish encrypted connections to the corresponding server.

3.2 Communication Modes

The communication between SMARTPROXY running on the smartphone and the web browser running on the untrusted computer is possible in several different ways:

Computer acting as WiFi access point: A computer provides a wireless hotspot or an ad-hoc wireless network to which the smartphone connects to. The smartphone's IP is displayed in the user interface and it is this address that is used as the proxy address by the web browser running on the untrusted machine. With this setup, all network traffic is "routed back" to the computer and this particular network connection is used for tunneling, as it is likely faster than the network connection available via GSM or 3G networks. Practically, solely rooted Android devices are able to connect to ad-hoc networks because vanilla Android devices do not support this connection mode.

Smartphone acting as WiFi access point: The Android OS is capable of serving as an access point for WiFi-enabled computers. For example, a notebook could connect to the smartphone's access point and utilize the proxy running on the phone. In this configuration, all traffic is routed over the smartphone's own Internet connection (*e. g.*, GSM, 3G, but not WiFi because it acts as an AP), and the linked-in computer will not

be able to observe *any* altered outgoing Internet traffic.

USB cable with USB tethering: In this setup, the smartphone is connected to the PC via the USB port. The phone needs to be configured for the USB tethering and the USB network device on the PC will in this case get an IP address from the smartphone. The smartphone's IP address can again be displayed within the user interface and is also used as the proxy address in the web browser running on the computer. This configuration routes all traffic through the smartphone's own Internet connection and the computer does not observe any additional network traffic (similar to the previous case, but this time the WiFi connection may be used). To the best of our knowledge, iOS does not support USB tethering on its own and Windows requires the smartphone's drivers to be properly installed. At the same time, however, most Linux distributions offer this functionality out-of-the-box.

Smartphone and Computer on same network: If both the computer and the smartphone use the same (wireless) network, then the smartphone is accessible from the computer and the computer can easily use the smartphone as a proxy. Furthermore, the smartphone may use the Internet connection offered by the WiFi network, instead of it's own GSM or 3G connection. If this setup is possible, it is the easiest to use in practice.

Upon familiarizing oneself with the list of communication methods above, one may notice that all these setups do not require any special access rights on either side. Although the first three setups require special privileges on the computer, they are likely to be enabled for all users because setting up a network interface is a common use case for most consumer targeted operating systems. This was an implementation aim for the software to be kept usable in most environments.

3.3 Proxy Workflow

We now present SMARTPROXY's workflow and the different steps that are necessary to enable the filtering of sensitive data. During the preparation stage, we need to connect the smartphone to the computer by using one of the methods discussed in Section 3.2. Next, it is necessary to set up the browser on the computer (*i. e.*, configure a proxy within the browser) and start the proxy software on the phone. If this is the first time SMARTPROXY is used, the tool's X509 V1 root certificate must be imported into the web browser. After performing these actions, the actual workflow can start (see Figure 1), which we discuss in the following.

1. SMARTPROXY listens for new network connections.
2. When the user opens a website in the browser on the computer, SMARTPROXY accepts this connection and spawns a new thread which parses the HTTP `CONNECT` statement.

3. SMARTPROXY opens a TCP connection to the desired web server, initiates an SSL handshake (called *target SSL handshake*) and verifies the server's X509 certificate. If it is not trusted or invalid (*e. g.*, the certificate is expired or the attacker on the

compromised computer attempts to intercept the connection), the tool ceases the action and notifies the user that an exceptional rule for this certificate has to be generated in the hopes of proceeding with the connection to the selected server.
4. If this is the first time a connection is established for this particular destination, we need to forge the supplied server certificate with our own RSA keypair and store it for later use. The forged certificate is an X509 V3 certificate signed by the proxy's CA certificate. Note that the user has imported this certificate in the browser in the preparatory phase, thus the browser regards this forged certificate as valid.
5. SMARTPROXY responds to the web browser with a plain text `200 OK` HTTP status code and upgrades the unencrypted TCP connection from the browser to an encrypted SSL connection with the forged certificate from the last step (called *local SSL handshake*). The web browser now assumes that it is talking securely with the designated web server. From this point forward, all network traffic between the web browser and the destination web server is encrypted and will be eavesdropped by the proxy. This and the following step are similar to a normal MitM attack except for the signing part. A more in-depth analysis of the MitM attack we have performed is available in Section 3.4.
6. After the connection setup is done, all requests from the web browser are served and filtered between the two endpoints. The filtering performed by our tool substitutes fake credential user data entered on the (potentially compromised) web browser with genuine credentials. Furthermore, we remove cookies and other types of sensitive data whilst hiding them from the web browser, as explained in Section 3.5. In order to provide some feedback to the user as to when user credentials are replaced, the smartphone vibrates and generates a visual output on each substitution.

This workflow corresponds to the SSL-secured HTTP connections, but a very similar approach can be used for plain HTTP connections with only minor modifications. Steps 3–5 are not needed, and solely standard HTTP requests (*e. g.*, `GET` or `POST` operations) have to be processed. We discuss the security implications of plain HTTP requests in Section 5.

3.4 Man-In-The-Middle Attack

Our approach relies on a MitM attack for the SSL-secured connections. Under the assumption that the user's computer is compromised, we need to presume that an attacker has total control over the machine and therefore can read and manipulate all data, including the input received from the proxy. This indicates that the SSL connection between the web browser and the proxy after the initial `CONNECT` statement may also be intercepted and is therefore somehow useless. To keep the confidential data hidden from the compromised machine, no encryption between the web browser and the proxy is required, as sensitive data is only transferred in another SSL-secured connection between the proxy and the designated web server. This connection can be guaranteed to remain unreadable by the attacker, even if the traffic is back-routed over the compromised computer. This is ascertained by the verification of the web server's SSL certificate within the SSL connection between the proxy on the smartphone and the target web server.

All these circumstances make it feasible to use no encryption on the connection between the web browser and the proxy, which would in turn decrease the security-pressure placed on the smartphone. However, this is unfortunately not supported by the web browsers we have tested. They instead expect an SSL handshake after each `CONNECT` statement and an SSL "downgrade" to the NULL cipher is not supported. All things considered, this behavior is clearly highly beneficial for general security reasons.

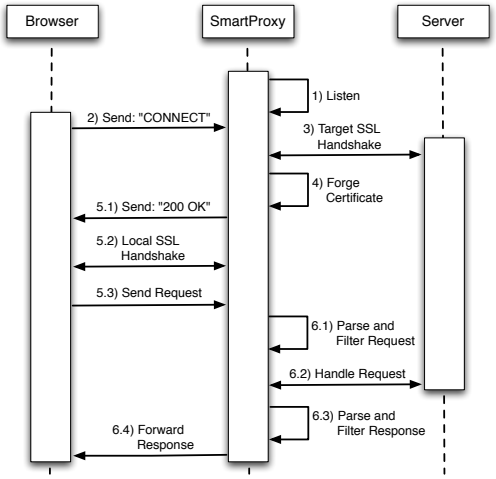

Fig. 1. Protocol sequence diagram.

In order to successfully carry out our MitM attack, we need two RSA keypairs. These keys have a modulus of 1024 bit which is believed to be safe enough, if we reckon the security point of view as the one elaborated on above. We did not choose stronger keys because of our desire to require as little as possible computational power on the smartphone. One of these RSA keypairs is used in an X509 V1 certificate, which is the root certificate of our own certificate authority. The other keypair is used within all forged X509 V3 certificates as the public key. This way, we only need to create two different RSA keypairs and helpfully save computational time when we have to forge a new certificate. The web browsers do not check whether all certificates they see use the *same* public key, thus they are prone to accept them as long as they have a valid digital signature (from our V1 certificate): the browsers verify the complete certificate chain, but use the same keypair for all relevant SSL operations. Interestingly enough, we actually have not intended doing this check-up in our implementation, but discovered it during the testing phase.

In order to keep the number of forged certificates low, hence enabling faster SSL Session resuming (see Section 4.1), step 5 from Section 3.3 generates forged certificates valid for more hosts than the original ones. If an original certificate is only legitimate for the host `a.b.c`, its new counterpart will be valid for the following hosts: `*.a.b.c`, `a.b.c`, `*.b.c`, and `b.c`. These *alternate subject names* are added to the forged certificate for each host which is found in the original one.

If the proxy is accessed over an insecure WiFi connection, the aforementioned encryption between the proxy and the web browser is suddenly a sound cause for concern. Another skilled attacker might sniff this connection and attempt to break the SSL encryption. This would allow her to observe the connection between the web browser and the proxy over the airlink. Such an attacker might not be detected by either side, thus the attacker in question might be able to capture data. However, this secondary attacker does not obtain more data compared to the attacker who has compromised the machine,

since both can only observe obfuscated data *after* the substitution by SMARTPROXY took place. An attack on the air link can be easily prevented by increasing the keysize to a strength of 2048 bits or more. It is also possible to generate and use different keys for each forged certificate.

3.5 Filtering

In order to transfer all security related information while surfing the web, the proxy has to change some elements in the data stream between the web browser and the web server. This is accomplished by implementing an extendable set of filters which can be attached to each data stream. These filters can in turn be used to change arbitrary HTTP header fields as well as payload data. When wishing to perform a login on a specific website without inserting the real credentials into the untrusted computer, the first action for a user to carry out is to set these accounts up within SMARTPROXY. The account data contains: real username and password, the domain where the login should be fulfilled, and fake username and password. All the above can be inserted into and modified through a GUI. The information is encrypted before it is written into a database (see Section 3.6 for details), and can then be used by our filters. Different filters are applied to the process of data modification, which we describe in the following.

Password Filter Real credentials should not be entered into the user's computer, therefore one has to exercise fake data input sent from the web browser to the server. Certainly, these fake credentials have to be substituted with genuine ones expected by the web server for the login's fulfillment. The Password Filter's function is to find and replace these fake credentials in the data stream. To do just so, the filter searches the `POST` data for the false credentials, determines if the credentials are indeed entered into valid form fields, and then performs the substitution. Furthermore, credentials are only substituted if the domain in the request matches the stored domain in the database.

In order to defeat some attacks, two additional checks are performed before the real credentials are inserted. First, we check if the password was already requested a short amount of time before (*e. g.*, 15 minutes). After a valid login, the user should be identified by some session ID and no credentials should be used anymore, thus this case might indicate an attack. Second, we check if the request contains at least three variables from which two are identical and a fake password. This could mean that someone is attempting to change the password (which might be an attacker, see Section 5). In both cases, SMARTPROXY asks the user for confirmation.

Authentication Filter To support HTTP authentication, we added a second filter that searches the `Authorization` field inside the request's header. In this case, we used the same framework as for the Password Filter, seeing as the user is also required to set up the corresponding fake credentials. Currently, our prototype supports the widespread *Basic Authentication* exclusively and not the lesser used but more secure *Digest Authentication*. To effectuate this functionality, we only had to substitute fake credentials with their genuine counterparts, just as we perform this in the Password Filter (except for them being Base64-encoded in this particular instance). The filter first decodes the

data, then checks for a positive match, substitutes the fake credentials, and finally encodes them back before performing the act of sending them to the server. Note that this filter cannot make use of the rate limit for substitutions as the Password Filter since each request will contain the fake credentials which have to be substituted.

Cookie Filter A potential attack against our approach could be a session hijacking attack through cross-site scripting. This event involves a session identifier (which authenticates the user in the eyes of the web server) that is stored in a cookie and can be stolen by an adversary, who can then employ this cookie to take over a complete session. To solve such issues, we modify the cookies sent through our proxy. Cookies are set by the web server with the `Set-Cookie` field in the HTTP header and are sent from the web browser to the web server with the `Cookie` field in the HTTP header. The first field is originating from a web server and has to be altered in such a manner that the data reaching the browser cannot be brought into operation for an attempted session's theft.

This is accomplished by all cookie properties (*e. g.*, VALUE, DOMAIN, NAME, PATH) being stored in a database and the value of cookies being changed to an arbitrary string. For the reverse direction (web browser to web server), the cookie values have to be restored from the database. Lastly, cookies can be created on the client machine directly in the browser with the use of techniques such as JavaScript or Adobe Flash. We decided to let these unknown cookies (for our proxy failed to notice a corresponding `Set-Cookie` HTTP header field) pass through in an unaltered stage because we do not think that an adversary can remodel such "self generated" cookies to be harmful for the user, and to be compatible with websites which depend on this behavior. Once we have dropped them, several websites responded and complained with warnings such as "*Your browser does not support cookies!*".

If we substitute all cookies, we might provoke errors with scripts which use cookie values to, *e. g.*, personalize the website. To overcome this, we only substitute cookies which have a value which is at least eight bytes long and has a high entropy since these cookies likely store sensitive data. Shorter cookies or cookies with low entropy are likely not security sensitive since an attacker could brute-force such cookies. Furthermore, all cookies which contain special strings such as *id*, *sid*, or *session* in the name are forced to be substituted. We assume that all security relevant cookies are protected this way and that personalized website settings are still functional. To be able to ensure the proper working of a website, a black- and a whitelist can be set up for each domain to define which cookies shall (not) be protected by SMARTPROXY.

Since some websites use cookie values to form special requests, we implemented a companion filter for the Cookie Filter to preserve the functionality of websites which depend on this behavior. One example is the Google Chat within GMail, which performs the following request: `GET /mail/channel/test?VER=8&at=COOKIE&i...` Here, the substituted value is meaningless and it hinders the website from operating normally. The purpose of the companion filter is to search for substituted cookie values in requests and to insert the correct value within outgoing requests.

Note that all cookies are only replaced within requests or the `COOKIE` header fields if the destination domain matches the domain which is stored in the database for the corresponding cookie. This further hinders an attacker from stealing cookies.

3.6 Personal Data Encryption

The smartphone is a kind of a single point of failure when one is investigating the secrecy of the stored user credentials. If it gets lost, stolen, or compromised, all deposited credentials must be considered known by a third party. To prevent this from occurring, we store all credentials in an encrypted form in the database in such a way that makes them unlikely to be revealed to a potential attacker. Each genuine username u_g and password p_g is encrypted with a key k derived from a fake password p_f using a *Password-Based Key Derivation Function* (PBKDF2 [18]). The encryption of u_g and p_g is performed with AES in CBC mode.

To improve the security of the genuine data, each fake password should be different. The fact that a fake password is used to derive the encryption key k for a real username and password leads to the fact that the fake passwords must not be stored in the database. This hinders the filters from Section 3.5 to find and replace these strings, as they may then appear almost anywhere in the header or payload, and their appearance may be just as arbitrary. To enable the matching of arbitrary fake password strings, the user enters the fake password in a special format, *e. g.*, `fp_fakepass1_`. That way, the filters have only to look for strings enclosed by a leading `fp_` and a closing `_`. This is a process that can be put in action easily. The intermediate string is used as p_f, and u_g and p_g may be decrypted with the latter operation's result. If by chance multiple strings are enclosed in our chosen markers, the corresponding filter will attempt to decrypt u_g and p_g with all found strings as p_f, eventually decrypting the credentials. The correct p_f can easily be determined by the use of a checksum.

4 Evaluation

We have implemented a fully-working prototype of SMARTPROXY. In this section, we present benchmark results, analyze the overhead imposed by our proxy software, and discuss some test results of using our proxy on popular websites. All benchmarks are divided into two categories, namely micro- and macro-benchmarks. The former measure "atomic operations' (*e. g.*, an SSL handshakes on the smartphone), while the latter are executed to estimate additional load time caused by our tool when accessing several popular websites. The smartphone we used in the testing phase was an *HTC Desire* with *Cyanogen Mod 7.0.3* (Android 2.3.3, not overclocked, using the default CPU governor). Additionally, the same benchmarks were performed on a vanilla *Samsung Nexus S* with Android 2.3.4. All results were comparable, thus we omit them in the following for conciseness reasons. On the client's side, we used *wget* in version 1.12 to automatically establish new connections, and our test smartphone device was connected to the Internet over a wireless connection to a 54 Mbit AP where a downstream of 2.7 MB/s is reachable. We used the `adb` port forwarding technique to launch and maintain a connection between the web browser and the proxy.

Table 1. Micro benchmark for the target SSL handshake (KS = Keysize in bit, AVG = Average Time, SD = Standard Deviation).

KS	Ciphersuite	AVG [ms]	SD [ms]
512	RSA/AES/256/SHA	29	24
1024	RSA/AES/256/SHA	33	17
2048	RSA/AES/256/SHA	37	9
4096	RSA/AES/256/SHA	90	17
512	DHE/AES/256/SHA	84	15
1024	DHE/AES/256/SHA	83	17
2048	DHE/AES/256/SHA	90	17
4096	DHE/AES/256/SHA	124	17

Table 2. Micro benchmark for the local SSL handshake (KS = Keysize in bit, AVG = Average Time, SD = Standard Deviation).

KS	Ciphersuite	AVG [ms]	SD [ms]
512	RSA/AES/256/SHA	35	16
1024	RSA/AES/256/SHA	42	20
2048	RSA/AES/256/SHA	90	68
4096	RSA/AES/256/SHA	360	326
512	DHE/AES/256/SHA	3,734	4,422
1024	DHE/AES/256/SHA	3,344	4,096
2048	DHE/AES/256/SHA	3,551	4,101
4096	DHE/AES/256/SHA	3,670	4,115

4.1 Synthetic Benchmarks

A main feature of the proxy is its ability to alter the payload in the SSL-secured HTTP connections, thus we need to evaluate the performance of the necessary steps. All measurement times are averaged over 100 test runs. In order to be able to measure these times without too much interference from the network latency, web server load, DNS lookups and so on, we have applied the benchmarks to a local server which is only three network hops away. The server is equipped with a Core i7 CPU, 8GB of RAM and was running Ubuntu 11.04 with Apache 2.2.17 using mod_ssl without any special configurations or optimizations.

SMARTPROXY is required to complete two different SSL handshakes for normal operation, hence we tested several configurations for this prerequisite. SSL Session resumption—for an explanation read below—was explicitly disabled for these benchmarks. At first, we tested the speed rate of the handshake to the target server, and evaluated how this depends on different RSA key sizes and ciphersuites. Then we repeated this test for the local SSL handshake. All measurement times are stated without the added-time necessary to validate a certificate, with the exception of a special check designed to investigate if the certificate has already been validated at some prior time. Certificates are validated when they are seen for the first time and then stored for the later use. Each "new" certificate is first checked against those pre-existing ones to determine whether it is already known and valid, and then, provided it is in the clear, it is accepted. This greatly speeds up the process thanks to the fact that the Online Certificate Status Protocol (OCSP) and certificate checks against Certificate Revocation Lists (CRL) are usually time-consuming, which needs to be done only once for each certificate during the proxy's runtime.

The first SSL-related action that occurs when a browser requests a secure connection to a HTTP server is a client handshake from the proxy to the designated host, necessary to establish a MitM controlled connection between these devices. As per our tests, this takes approximately 33 ms for a 1024 bit strong RSA key with our own server. Table 1 provides an overview of the benchmarking results for establishing a secure channel between the proxy and our local test web server, all for different setups. What we discuss below is that these handshakes were all completed in an acceptable time, even with a

4096 bit strong RSA key, and, equally, for the ephemeral Diffie-Hellman (EDH or DHE) ciphersuites, which are generally slower.

In the second step, the certificate presented by the web server has to be forged in order to establish a secure and trusted connection between the web browser and the proxy. The average time to forge and sign such a certificate is 150 ms, which is sufficiently fast and needs to be only performed once for each accepted certificate. Operations like persisting a new certificate in a keystore are also here-contained.

During the third and final step, the web browser upgrades its plain text connection to the proxy to an SSL-secured connection-type. This implies an SSL server handshake on the proxy involving a forged certificate. Once again, this is a pretty fast operation which takes only 42 ms on average for a 1024 bit strong RSA key. Table 2 provides an overview of how long the local SSL handshake takes for several configurations. In order to accomplish decent times for the local handshakes, it is crucial to choose the right ciphersuites. While plain RSA handshakes operate fast enough even for a 2048 bit key, a noticeable delay is caused by a 4096 bit key. Conversely, if the web browser chooses EDH ciphersuites for the session key exchange, the time to complete such a handshake might be pinned down as anywhere from 1 to up to 10 or more seconds for each handshake—even with the same certificate, which explains the high standard derivation for these tests. EDH handshakes are more secure, yet evidently and understandably more expensive than plain RSA handshakes. Still, normally they do not reach the factor of 100 which we measured. We have reasons to believe that this operation is not well optimized in the used *OpenSSL* stack on Android, albeit it employs the native code in order to do the expensive operations. The fact that SSL handshakes with EDH ciphersuites to the target server are much faster indicates that this code lacks optimizations for the server mode of the handshake, cf. Section 4.2. We believe that it takes a relatively long time to compute the EDH parameters and this phenomenon would explain the vast amount of time it takes to complete a local handshake with EDH ciphersuites. Conclusively, to enable fast local SSL handshakes, we only accept the following secure ciphersuites for these cases: `RSA/RC4/128/SHA`, `RSA/RC4/128/MD5`, `RSA/AES/128/CBC/SHA`, and `RSA/AES/256/CBC/SHA`. Nevertheless, all available secure (EDH) ciphersuites are enabled for the SSL handshake to the web server.

In order to accelerate the expensive SSL handshakes, SSL and TLS support the mechanisms called *Sessions*. Sessions are identified with a unique ID by the client and the server. Once a completed handshake is bound to such a Session, it will not happen again when the Session is still valid. The client and the server may both reuse a Session, and the already negotiated cryptographic parameters that come with it. This feature can greatly facilitate the establishment of new connections. As an initial handshake from the first step is completed, a new connection to our web server is established within under 30 ms on average from the proxy. A standard estimate for a resumed local server handshake to take place is about 40 ms.

We did not benchmark the SSL operations for different websites separately because all operations, except the SSL handshake to the web server, use the same static key materials in the first step which results in almost the same benchmark for each operation.

Table 3. Evaluation results for global Top 25 websites from alexa.com without duplicates and sites that do not support SSL (KS = Keysize, HS = Handshake, SD = Standard Deviation, LT = Load Time, OH = Overhead).

	Micro Benchmarks				Macro Benchmarks		
website	KS [bit]	Ciphersuite[a]	HS [ms]	SD [ms]	LT [s]	OH	Login
google.com	1024	RSA/RC4/128/SHA	91	49	0.95	23%	✓
facebook.com	1024	RSA/RC4/128/MD5	424	103	1.37	58%	✓
youtube.com	1024	RSA/RC4/128/SHA	71	8	2.93	20%	✓
yahoo.com	1024	RSA/RC4/128/MD5	192	52	4.34	26%	✓
wikipedia.org	1024	RSA/RC4/128/MD5	363	169	5.11	50%	✓
live.com	2048	RSA/AES/128/SHA	595	31	1.60	23%	✓
twitter.com	2048	RSA/RC4/128/MD5	400	97	3.66	17%	✓
linkedin.com	2048	RSA/RC4/128/MD5	426	72	1.93	78%	✓
taobao.com	1024	RSA/RC4/128/MD5	2,716	2,817	34.82	154%	↯[b]
amazon.com	1024	RSA/RC4/128/MD5	263	19	2.24	18%	✓
wordpress.com	1024	DHE/AES/256/SHA	527	52	16.66	204%	✓
yandex.ru	1024	DHE/AES/256/SHA	274	50	5.75	260%	✓
ebay.com	2048	RSA/RC4/128/MD5	587	51	1.49	46%	✓
bing.com	1024	RSA/RC4/128/MD5	52	25	0.62	142%	✓

[a] DHE with RSA and AES in CBC mode.
[b] We were not able to create a login for this website because we could not translate the content of this site.

4.2 Real-World Benchmarks

We have selected the global top 25 websites as they are listed at `www.alexa.com` to be our test sites for additional benchmarking, yet we omitted "duplicate" sites like `google.co.jp` and `google.de`, as well as all the sites we were unable to understand content-wise due to its language (despite the attempted use of website translators), *e. g.,* `qq.com`. All the web pages tested are listed in Table 3. The main feature of the proxy is its ability to withhold crucial information such as user credentials and cookies, cf. Section 2. This functionality is tested on all popular websites. Whenever it was possible, we created a login and tested the functionality. We found out that our proxy successfully substituted fake credentials with genuine ones on all tested websites.

We measured SSL handshakes for real-world sites (micro-benchmarks) and again used *wget* for these tests. *Firefox* 5.0 with the *Firebug* 1.8.1 plugin was used to measure website load times (macro-benchmarks). No caching was undertaken between each page reload and we additionally invalidated all SSL Sessions after each connection for the micro-benchmarks. The certificates are validated by mirroring the method described in the previous section. Table 3 lists all times and the implied overhead.

To keep the impact of unknown side effects low, we trialled all websites in a row and repeated this 100 times on a rolling basis. In order to measure the overall overhead (macro-benchmarks), we tested each website five times and calculated the load time, comparing the results for when the proxy was used versus the same action without it.

It is vital to note that these times include a lot of operations with timings which we cannot influence or control. These include the web servers load, the time needed to look up the IP address, certificate checks (OCSP/CRL), as well as general network latency. Overall, the SSL handshake from our proxy to the web server is fast enough not to hinder the user's experience. This is interesting because the SSL Sessions can be resumed, the feature which was explicitly disabled for the micro-benchmarks testing here (see Section 4.1). As this is the time it takes for the initial handshake, it does not have a huge impact on the page load times for the subsequent handshakes. One notable exception is `taobao.com`, but this page was generally slow on all tested devices.

Table 3 shows that the total overhead is reasonable when SMARTPROXY is used as a privacy proxy. While the tool has a certain overhead, it does not in any case makes it unacceptable to surf the Web. For the majority of the tested sites, the overhead is less than 50%, which means that the web page will load up to 1.5 times slower. These times refer to a complete web page load with an empty cache. When the web page is accessed with the proxy and the web browser uses its cache, the overhead is sometimes not even noticeable by a user. Some sites such as `twitter.com` and `yandex.ru` have a larger infrastructure and several SSL handshakes may be needed in order to load a single page, for the content is served from many different hosts with different SSL certificates. Our approach to add additionally derived alternate subject names to the forged certificates, cf. Section 3.4, may reduce the amount of SSL handshakes which are needed to be performed, although this depends on the infrastructure of the specific website, as the completely different overhead for these two web pages shows in Table 3.

Another aspect which has a certain impact on the overhead introduced by SMARTPROXY is the method used by the web servers to handle the connections. If they close a connection rather than allowing our tool to reuse it, we need to establish a new connection to the proxy and to the web server, which implies *two* additional SSL handshakes. This takes time and is clearly visible in the measurement results for `wordpress.com`, a site that exhibits this behavior. This is obviously an implementation detail, but our prototype currently allows only "pairs of connections": namely one connection established from the web browser is tied to one connection to a designated web server.

Aside for "normal web surfing", we evaluated how fast we are able to download large files. The average download speed with SMARTPROXY enabled is only slightly slower than the download without the proxy. We have seen download rates that differ by only a few KB/s up to some 100 KB/s. This of course depends on the accessed server and the WiFi connection's quality. The fastest download speed achieved with the proxy enabled was 2.7 MB/s. To complete our evaluation, we did some research into a selection of the video portals to check if they work correctly. The video portals typically make use of many different web technologies to stream the video content to the browser. When performing our tests, we have not identified any problems on the three most popular video portals from the global Alexa ranking.

5 Limitations

We now discuss several limitations of our solution and at the same time also sketch potential future improvements to strengthen SMARTPROXY. As already explained in

Section 3.1, the smartphone may route all its traffic back to the potentially compromised computer, as this is the machine engaged as the Internet access point (default gateway). Clearly, this poses a security problem in case that the web browser uses the proxy for plain HTTP websites. Because all HTTP traffic is unencrypted, an attacker may read and alter all traffic on the compromised computer. This includes all data that have been formerly replaced; or, in other words, the genuine credentials and cookies. The current implementation does not check for this scenario. However, all SSL-secured connections are not affected because a MitM attack would be detected by the proxy running on the smartphone—as long as the PKI is trustworthy. Furthermore, we could extend SMARTPROXY to automatically "upgrade" each HTTP to a HTTPS connection whenever possible, thus preventing this attack scenario. Another potential weakness is that the login procedure on a website could perform some kind of computation during the login process with techniques such as JavaScript, Java, Adobe Flash, Active-X or other similar technologies. These procedures could compute arbitrary values which are in turn sent to some unknown destinations. Currently, we do not support these setups.

Our cookie filtering approach might also cause trouble on some websites due to over- or underfiltering. We have not found any problems during our tests and evaluation but can not guarantee that it always works as expected. Some websites might require the use of the white- and blacklist feature.

As the attacker is able to interact with all the content of a web site, she is able to perform so called *Transaction Generator Attacks* [16]. This means that she can make all actions on behalf of the user, *e. g.*, make orders in web stores, post messages, and so on. This cannot effectively be prevented by SMARTPROXY, as it cannot tell by any means which requests are legit and which are not. Jackson *et al.* propose some countermeasures [16], which could be included in SMARTPROXY. The best defense SMARTPROXY offers for now is that the user can always see what is going on as all requests are visible on the smartphone's screen. The user might simply cut the connection to the proxy if something looks suspicious. In this case, the attacker might disable the usage of the proxy, but will not be able to proceed, as no credentials and cookies are available to her.

We did not carry out a user study to see how successful people are able to set up SMARTPROXY. We are aware that people often have difficulties understanding what is going on with security related problems (see for example [6]). As long as the users remember to never input real credentials into the computer while they use SMARTPROXY, they can at least be assured that their credentials are safe.

Finally, SMARTPROXY is not yet really scalable. Each web browser which shall be used with the proxy has to be set up. This includes importing the root certificate for SSL secured websites. Albeit the certificate is not crucial if the link between the smartphone and the computer is secure—*e. g.*, when connected with the USB cable—, it greatly enhances the user experience as otherwise each SSL secured website will enforce a certificate warning from the browser.

6 Related Work

Research in this area to date was focused on the attempts to solve similar problems as we discuss in the following. Mannan and Oorschot [22] proposed an approach to secure

banking transactions by using a (trusted) smartphone as an additional trust anchor. In a very similar approach, Bodson *et al.* [7] proposed an authentication scheme where users have to take pictures of QR codes presented on websites in order to log in securely. The picture taken is then sent over the smartphone's airlink to the web server for the authentication to be performed there externally. To fulfill their needs, both approaches had to implement server side changes. Conversely, one of our main goals is to only alter the client's side by adding a self-signed root certificate to a browser and configuring the smartphone as a HTTP/HTTPS proxy.

Hallsteinsen *et al.* [13] also used a mobile phone to obtain a unified authentication. This approach is based on one-time passwords on the one hand and the need for a working GSM system on the other. In this research, an additional Authentication Server is connected to both the GSM network and the service provider which the user wants to authenticate against. The authentication requests are sent via short messages (SMS) in this setup. By doing this, the authors mitigate the risk of MitM attacks, but have to trust the safety of the user's computer, which in our scenario might as well be compromised.

Balfanz and Felten [3] developed an application that moves the computation of the e-mail signatures to a trusted mobile device so that the untrusted computer can be used without revealing the key to the intermediary computer. However, this approach is limited in the application scope.

Perhaps closest to our idea is *Janus*, a Personalized Web Anonymizer [10]. The idea behind Janus is to enable simple, secure and anonymous web browsing. Credentials and email addresses are automatically and securely generated with the help of a proxy server for HTTP connections. The proxy assures that only secure credentials and no identifying usernames and email addresses are sent to the web server. In contrast to our work, Janus does only support SSL secured connections from the proxy to the web server and the proxy resides on the client machine, which does not fit our attacker model. It does also not handle other security relevant information like cookies. Finally, Ross *et al.* [24] proposed a browser plugin which automatically generates secure login passwords for the user. The user has to enter a "fake password" which is prefixed with `@@` and which is replaced by the plugin by some secure password. This approach is similar to our credential substitution process, but does not provide a remedy against compromised machines.

7 Conclusion and Future Work

In this paper we introduced an approach to protect user credentials from an eavesdropper by taking advantage of a smartphone that acts as a privacy proxy, *i. e.*, the smartphone transparently substitutes fake information entered on an untrusted machine and replaces it with genuine credentials. We showed that it is possible to enable secure logins despite working on a compromised machine. The steps for achieving the desirable output are relatively easy, as they mainly require connecting the smartphone to the computer and configuring the web browser so that it uses our proxy. Compared to previous work in this area, we do not need a trusted third party that performs the substitution, but everything is handled by the smartphone itself. We have implemented a fully-working prototype of our idea in a tool called SMARTPROXY. The overhead is reasonable and

often even unnoticeable to the user as demonstrated with our benchmarking results. Furthermore, we evaluated the security implications of the setup and we are convinced that our solution is beneficial for many users.

In the following, we briefly discuss future work. First, there could be a better connectivity between a smartphone and a computer. A type of an automatic wireless connection would be helpful, and the use could potentially be made of the Bluetooth protocol. We did not look into this aspect because the current connectivity scope was deemed sufficient for the prototype. It would be advantageous to see some kind of "Reverse Tethering" support on Android, like the ActiveSync feature that Windows Mobile offers. This could enable effortless use of the fast Internet connection of the computer. Furthermore, a guarantee of a proper detection of cases when the plain HTTP traffic is routed back to the computer on which the web browser runs would be essential in order to prohibit vulnerable setups of this sort.

Additional filters could be implemented to hide content from web browsers, for example by some type of blacklist to substitute valuable data with fake information. Examples could include names, addresses, credit card and social security numbers, and even real credentials. This would also avert "mirror attacks" where an attacker attempts to send substituted real data back to the web browser through the proxy to get hold of them. Although the attacker would need access to the web server to enable such mirroring and would already have access to the data, such an attack would be harder for the attacker. As the filter system is pluggable, this can be implemented rather easily but was not done for our prototype at this stage.

Last but not least, the speed of the SSL handshakes could be improved. Although it is already pretty fast for most use cases, this is undoubtedly a desirable direction of improvements in the future. There are some existing solutions which are known to reflect and cater to this need, but they are currently unavailable on the Android OS. One example would be what Google proposes in the instances of *False Start* [21], *Next Protocol Negotiation* [19], *SPDY* [1] and *Snap Start* [20]. These solutions are currently tested by Google in selected applications and servers. If they prove successful, they will be hopefully integrated into Google's Android.

Acknowledgments This work has been supported by the Federal Ministry of Education and Research (grant 01BY1020 – MobWorm), and the DFG (Emmy Noether grant *Long Term Security*).

References

1. SPDY. `http://www.chromium.org/spdy`. Accessed 04.11.2011.
2. Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 49(14), 1996.
3. D. Balfanz and E. W. Felten. Hand-Held Computers Can Be Better Smart Cards. In *USENIX Security Symposium*, 1999.
4. M. Burnside, D. Clarke, B. Gassend, T. Kotwal, M. van Dijk, S. Devadas, and R. Rivest. The untrusted computer problem and camera-based authentication. In *In Pervasive Computing, volume 2414 of LNCS*, pages 114–124. Springer-Verlag, 2002.
5. C. Drew (The New York Times). Stolen Data Is Tracked to Hacking at Lockheed. `http://www.nytimes.com/2011/06/04/technology/04security.html`. Accessed 04.11.2011.
6. S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *USENIX Security Symposium*, 2006.

7. B. Dodson, D. Sengupta, D. Boneh, and M. S. Lam. Secure, Consumer-Friendly Web Authentication and Payments with a Phone. In *Proceedings of the Second International ICST Conference on Mobile Computing, Applications, and Services (MobiCASE)*, 2010.
8. EMC Corporation. RSA SecurID. `http://www.rsa.com/node.aspx?id=1156`. Accessed 04.11.2011.
9. J. Franklin, A. Perrig, V. Paxson, and S. Savage. An inquiry into the nature and causes of the wealth of internet miscreants. In *ACM Conference on Computer and Communications Security (CCS)*, 2007.
10. E. Gabber, P. B. Gibbons, Y. Matias, and A. Mayer. How to Make Personalized Web Browsing Simple, Secure, and Anonymous. In *Financial Cryptography and Data Security (FC)*, 1998.
11. gera. Advances in Format String Exploitation. *Phrack Magazine*, 59(12), 2002.
12. J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Commun. ACM*, 52:91–98, May 2009.
13. S. Hallsteinsen, I. Jorstad, and D. Van Thanh. Using the mobile phone as a security token for unified authentication. In *Proceedings of the Second International Conference on Systems and Networks Communications (ICSNC)*, 2007.
14. W. Henecka, A. May, and A. Meurer. Correcting Errors in RSA Private Keys. In *Advances in Cryptology (CRYPTO)*, 2010.
15. T. Holz, M. Engelberth, and F. C. Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.
16. C. Jackson, D. Boneh, and J. Mitchell. Transaction generators: root kits for web. In *USENIX Workshop on Hot Topics in Security (HotSec)*, 2007.
17. R. C. Jammalamadaka, T. W. van der Horst, S. Mehrotra, K. E. Seamons, and N. Venkasubramanian. Delegate: A Proxy Based Architecture for Secure Website Access from an Untrusted Machine. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.
18. B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. `http://tools.ietf.org/html/rfc2898`, 2000.
19. A. Langley. Transport Layer Security (TLS) Next Protocol Negotiation Extension. `https://tools.ietf.org/html/draft-agl-tls-nextprotoneg-00`, 2010.
20. A. Langley. Transport Layer Security (TLS) Snap Start. `http://tools.ietf.org/html/draft-agl-tls-snapstart-00`, 2010.
21. A. Langley, N. Modadugu, and B. Moeller. Transport Layer Security (TLS) False Start. `https://tools.ietf.org/html/draft-bmoeller-tls-falsestart-00`, 2010.
22. M. Mannan and P. C. Van Oorschot. Using a Personal Device to Strengthen Password Authentication From an Untrusted Computer. In *Financial Cryptography and Data Security (FC)*, 2007.
23. Nergal. The Advanced return-into-lib(c) Exploits: PaX Case Study. *Phrack Magazine*, 58(4), 2001.
24. B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *USENIX Security Symposium*, 2005.
25. H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.
26. M. Wu, S. Garfinkel, and R. Miller. Secure Web Authentication with Mobile Phones. In *DIMACS Workshop on Usable Privacy and Security Systems*, 2004.
27. Yubico. YubiKey - The key to the cloud. `http://www.yubico.com/products-250`. Accessed 04.11.2011.